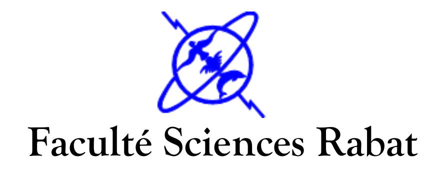

Manuel d'implémentation des Web Services Sous Axis1 et Axis2/Tomcat/linux

Par Pr Bouabid EL OUAHIDI

Email : ouahidi@fsr.ac.ma
https://sites.google.com/site/bouabidouahidi/

Ce document représente mes notes de cours sur le cloud computing et web services. Cet enseignement nécessite certains prérequis dont entre autres : Java, système linux, programmation répartie (socket, RPC et RMI). J'ai des documents (notes de mes cours sur la programmation répartie) que je peux vous envoyer en cas de besoins.

Cet enseignement s'adresse en priorité aux étudiants niveau licence SMI et Master informatique.
Très prochainement, je mettrai en ligne dans le cadre d'un MOOC de la faculté des sciences de Rabat, des vidéos sur la programmation répartie socket, RPC, RMI et les Webs Services.

Année 2013-2014

I) Le but est de développer des services Web avec des technologies Open Source.

Pour le déploiement des Web services, nous utilisons Tomcat (moteur de Servlet et de JSP) et Axis (une implémentation Java Open Source de SAOP). Des exemples de web services avec axis1 et axis2 seront développés. Vous aurez aussi à la fin de ce document des informations sur XML-RPC Java qui est une technologie moins «importante» pour développer des web services.

Je rappelle que la technologie des Web services se base sur:

1) SOAP (le protocole d'échange de messages XML (requêtes et réponses) entre client et serveur
2) WSDL (Web Service Description Langage) est un langage reposant sur XML dont on se sert pour décrire les services Web offerts pour la description en XML.
3) UDDI annuaire des descriptions WSDL.

II) Installation tomact et Axis

Cette partie concerne axi1, c'est-à-dire la version axis-1-4.

Je vous conseille donc de télécharger les versions binaires de tomcat et de axis à partir respectivement des sites: http://jakarta.apache.org, http://ws.apache.org/axis/ **(Mettre à jour)**

Mais vous pouvez trouver par google des sites miroirs plus rapides en cherchant axis et tomact.

II-A) Variables d'environnement

Un certain nombre de variables d'environnements doivent être initialisées avant toute utilisation de tomcat et de axis.
J'ai choisi d'utiliser un fichier shell script que je nomme config.sh. On aurait pu mettre ce script dans .bashrc.

Votre fichier config.sh doit ressembler à ceci :

```
# Vous trouvez plusieurs fois unset, car j'utilise x=$x :…., et en cas de
# lancement de plusieurs fois ce qui répète le contenu autant de fois.

# Pour JAVA
#  JAVA_HOME =repertoire de JDK
unset JAVA_HOME
# JAVA_HOME est le dossier d'installation de votre jdk
	JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.9
	export JAVA_HOME
#Pour tomact
#CATALINA_HOME=repertoire de tomcat
unset CALALINA_HOME
	CATALINA_HOME=/usr/local/apache-tomcat-7.0.33
	export CATALINA_HOME

#Pour axis1
```

```
#AXIS_HOME= repertoire d'axis1
unset AXIS_HOME
        AXIS_HOME=/usr/local/axis1
        export AXIS_HOME

#Pour Axis2
#AXIS2_HOME=repertoire axis2
unset AXIS2_HOME
        AXIS2_HOME=/usr/local/axis2
        export AXIS2_HOME

#PATH
# faire d'abord echo $PATH et copier son contenu.
unset PATH
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/home/ouahidi
/.local/bin:/home/ouahidi/bin
        PATH=${PATH}:${HOME}/bin:.
        PATH=${PATH}:${CATALINA_HOME}/bin
        PATH=${PATH}:${JAVA_HOME}/bin
        PATH=${PATH}:${AXIS2_HOME}/bin

        export PATH


# Pour XMLRPC
unset XMLRPC_HOME
XMLRPC_HOME=/usr/local/apache-xmlrpc-3.1.1
export XMLRPC_HOME
```

La variable CLASSPATH est importante, mettre les les .jar dedans.
Ici à titre indicatif. Enlever évidemment le #.

```
#CLASSPATH
#unset CLASSPATH
# Pour XmlRpc et le .
#CLASSPATH=${XMLRPC_HOME}/commons-logging-1.1.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-common-3.1.1-
javadoc.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/ws-commons-util-
1.0.2.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-common-3.1.1-
sources.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-client-3.1.1.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-server-
3.1.1.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-client-3.1.1-
javadoc.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-server-3.1.1-
```

```
javadoc.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-client-3.1.1-
sources.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-server-3.1.1-
sources.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/xmlrpc-common-
3.1.1.jar


#CLASSPATH=${XMLRPC_HOME}/lib/commons-logging-1.1.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/lib/xmlrpc-client-
3.1.2.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/lib/xmlrpc-server-
3.1.2.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/lib/ws-commons-util-
1.0.2.jar
#CLASSPATH=${CLASSPATH}:${XMLRPC_HOME}/lib/xmlrpc-common-
3.1.2.jar



#
# Pour Axis
#unset CLASSPATH
# CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/axis.jar
# CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/jaxrpc.jar
#   CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/commons-discovery-
0.2.jar
# CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/saaj.jar
#      CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/commons-logging-
1.0.4.jar
# CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/log4j-1.2.8.jar
# CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/wsdl4j-1.5.1.jar
# CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/axis-ant.jar
# CLASSPATH=${CLASSPATH}:${AXIS_HOME}/lib/activation.jar
# CLASSPATH=${CLASSPATH}:${CATALINA_HOME}/webapps/axis

#CLASSPATH=${CLASSPATH}:/usr/share/java/activation.jar:.


# export CLASSPATH




# CLASSPATH=${CLASSPATH}:/usr/xerces-2_5_0/xerces.jar
#CLASSPATH=${CLASSPATH}:/usr/xerces-2_5_0/xml-apis.jar
```

#CLASSPATH=${CLASSPATH}:/usr/xerces-2_5_0/xmlParserAPIs.jar
#CLASSPATH=${CLASSPATH}:/usr/xerces-2_5_0/xercesSamples.jat

#Pour WSDL
#setenv CLASSPATH ${CLASSPATH}:${HOME}/WebServices/ExemplesWebServicesAxis/AvecWSDL

#setenv CLASSPATH ${CLASSPATH}:${MYSQL_HOME}/mysql-connector-java-3.0.15-ga-bin.jar

II-B) Lier Tomact et Axis

a) Copier le dossier axis
Pour lier tomcat et axis, copier le répertoire **axis** qui se trouve dans ${AXIS_HOME}/webapps dans ${CATALINA_HOME}/webapps/.

Ainsi le serveur axis est installé comme une application Web au sein de tomact.

Sous shell, il suffit de faire:

$ cp -Rf ${AXIS_HOME}/webapps/axis ${CATALINA_HOME}/webapps/

b) Tester l'installation
lancer tomcat comme suit : $ startup.sh
(ceci si vous l'aviez déjà ajoute dans le PATH sinon aller dans ${CATALINA_HOME}/bin)

Vérifier que tomcat tourne correctement dans un navigateur : http://localhost:8080/

Valider à présent l'installation d'axis en vérifiant que les paquetages obligatoires sont présents: http://localhost:8080/axis/
Ensuite, cliquer sur Validate

Le message suit doit apparaître **: « The core axis librairies are present »**

L'environnement est prêt pour le développement de services webs tomcat/axis sous linux.
Sous windows, vous aurez pratiquement les mêmes variables d'environnement, seule la syntaxe change.

III) Développement et déploiement de services Web

Comment déployer un service web à présent. Axis propose deux manières de déployer un service : en utilisant un fichier JWS et, en passant un fichier WSDD à `AdminClient`

La première approche est beaucoup plus simple que la seconde, mais ne permet pas la

personnalisation (ne pas exposer que certaines méthodes par exemple) du déploiement.

III-1) JWS (Java Web Service)

JWS est une technologie d'Apache permettant d'écrire rapidement un service web qui se déploie instantanément une fois la source mis dans un répertoire particulier.

On va créer un service Web qui affiche un message de bienvenue à l'utilisateur. Ce service expose une méthode appelée `sayHello()`. Cette méthode prend un argument `String` et retourne un message `String` au client.

Nous allons donc écrire une classe `HelloService.java`

```
public class HelloService {
     String sayHello(String argument)
     {return      "Hello "+argument ;
     }
```

Le développement du service Web est ainsi terminé : inutile de compiler ce code Java. Il suffit de copier `HelloService.java` dans ${CATALINA_HOME}/webapps/axis/ en le renommant `HelloService.jws`, **sans compilation!! Remarquez également qu'il n'y a pas de directives import !**

Sous shell, il suffit de faire la commande.
$ cp HelloService.java ${CATALINA_HOME}webapps/axis/HelloService.jws

C'est fini votre Web service est prêt et il est accessible avec l'URL:
http://localhost:8080/axis/HelloService.jws

Vous devez alors constater que votre service a été bien déployé sur "axis" en ayant en retour la page HTML suivante:
There is a Web Service Here
Click to see the WSDL

Si vous cliquez sur ce lien, vous voyez la définition "WSDL" (généré automatiquement par Axis de votre service Web). Si cette description n'apparaît pas (sur certains navigateurs), vous pouvez toujours la stocker localement et la visualiser.

La description WSDL est générée à la volée est maintenant accessible sous :

http://localhost:8080/axis/HelloService.jws?wsdl

Cette description est utile pour tout client souhaitant accéder au service en générant ou non les stubs nécessaires. Analyser cette description et retrouver votre méthode et les paramètres et le point d'accès (endpoint).

Tester également: http://localhost:8080/axis/index.html
Cliquer sur view et que voit-on apparaître ?

Avant de décrire la deuxième approche avec WSDD, la suite décrit la manière d'éxécuter un service web.

Exécution du service Web

- Sans développer de client Java

Votre service est accessible à travers de partout, il suffit d'avoir une connexion internet et un navigateur !!

Dans un premier temps, on va exécuter la méthode `sayHello`() sans écrire de client Java, mais on aura uniquement des réponses SOAP.

Pour exécuter cette méthode et obtenir une réponse "SOAP" correspondante, il suffit de mettre l'URL suivante dans votre navigateur:

http://localhost:8080/axis/HelloService.jws?method=sayHello

Et pour passer un argument (ici une chaine de caractères) à `SayHello`, faire:

http://localhost:8080/axis/HelloServic?method=sayHello&argument="GENIAL"

La réponse affichée est le contenu "SOAP", c'est à dire un fichier XML. Sur certains navigateurs vous verrez : Hello GENIAL

- Un client Java qui appelle notre service Web.

On verra par la suite qu'il existe plusieurs manières. Ici on ne se sert pas de la définition WSDL pour générer le client. Voici une plus simple.

```
import org.apache.axis.client.*;
import javax.xml.namespace.QName;

public class TestHelloService{
    public static void main(String[] args)

{ try {

  /* Tomcat est lance par startup.sh sur la machine locale
 Le port utilise est 8080 qui peut être change dans  server.xml
(voir     ${CATALINA_HOME}/conf/server.xml */

// L'URL d'accès au service  Web

String endpoint ="http://localhost:8080/axis/HelloService.jws";

/* On Crée un objet  service de la classe Service(). Cela
suppose que le client construit tous les paramètres et non pas
les prend à  partir de wsdl du serveur */

 Service service = new Service();

/* Construction d'un objet call permettant de construire la
requête et  réponse SOAP */

       Call call = (Call) service.createCall();

 // On crée un objet associé à  l'URL endpoint
call.setTargetEndpointAddress(new java.net.URL(endpoint));

// On précise la méthode à exécuter (ici on une seule)
```

```
call.setOperationName("sayHello");

//On invoque avec les paramètres (tableau d'Object) et on
//reçoit le résultat. Args[1] contient la chaine à passer à
//SayHello()

String resultat = (String) call.invoke(new Object[] {args[1]});
//
System.out.println(resultat);


} catch (Exception e) // pour simplifier
         {System.err.println(e.toString());
         }
     }
}
```

Compiler et exécuter.

```
$ javac TestHelloService.java
$ java TestHelloService GENIAL
```
Hello GENIAL

On verra d'autres manières d'écrire les clients. A présent construisons un autre exemple avec toujours la même manière de déploiement, c'est à dire avec un fichier JWS.

Ecrire une «super-calculatrice» ayant les méthodes somme(int,int) et produit(int,int) pour l'instant (soyons modeste :)

- Écriture du service web : il s'agit de votre fichier `CalculService.java` (écrit très simplement : aucun import, pas d'héritage, etc...)
- Déploiement : il suffit de copier le fichier `CalculService.java` dans le répertoire ${CATALINA_HOME}webapps/axis en le renommant en `CalculService.jws`

```
Public class CalculService{
Public int somme (int a, int b) {return a+b ;}
Public int produit(int a, int b){ return a*b ;}
}
```

Exécution à travers un navigateur :
Tester : http://localhost:8080/axis/CalculService.jws?wsdl

Analyser sa description WSDL et retrouver les méthodes métier, les paramètres, et le point d'accès (endpoint).

http://localhost:8080/axis/CalculService.jws?method=somme&a=3&b=5
http://localhost:8080/axis/CalculService.jws?method=produit&a=3&b=5

Développons un client Java

```
// Permet de tester CalculService.jws qui a deux  méthodes
somme et produit.
// Ce fichier est mis dans
${CATALINA_HOME}/webapps/axis/CalculService.jws
// Attention à l'extension jws et non pas java

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;

public class TestCalculService{
  public static void main(String[] args)
{ try {

     // enpoint est l'URL du serveur Web
     String                                            endpoint
="http://localhost:8080/axis/CalculService.jws";

          Service service = new Service();
          Call call = (Call) service.createCall();
          call.setTargetEndpointAddress(new
     java.net.URL(endpoint));

           // On précise la méthode à exécuter est somme
              call.setOperationName("somme");
 // On invoque avec les paramètres (tableau d'Objet)  et on
// Reçoit le résultat.

//les deux entiers
           Integer i = new Integer(5);
           Integer j = new Integer(7);


           Integer  ret  =  (Integer)  call.invoke(new  Object[]
{i,j});
            System.out.println("Somme = "+ret.intValue());

// l'autre méthode si l'on veut
// On précise la méthode a exécuter est produit
call.setOperationName("produit");

// On invoke
ret = (Integer) call.invoke(new Object[] {i,j});

System.out.println("produit = "+ret.intValue());
= "+ret.intValue());
} catch (Exception e)
 { System.err.println(e.toString());
```

```
        }
}
}
```

Il à noter que ce code peut être simplifié en utilisant Java 7.

III-2) Approche WSDD

On va maintenant aborder une autre approche de déploiement des services web. Le fichier JWS offre le moyen le plus rapide et le plus simple de déployer un service Web sous Axis. Il ne permet cependant pas de personnaliser le déploiement. En outre, il laisse le code source sur le serveur à disposition d'éventuels pirates et ne permet pas de décider d'une expression sélective de certaines méthodes. Pour cela, Axis propose une manière de personnaliser le déploiement: WSDD (Web Service Delloyment Descriptor), le descripteur de déploiement de service Web.

Le WSDD est un document XML (toujours lui). Il contient les informations sur le service Web à déployer et à spécifier le nom de la classe du service Web, la liste des méthodes à exposer, etc.

Voici les étapes:

<u>a) Ecrire un fichier.wsdd</u>

On compile `HelloService.java` et on met la classe `HelloService.class` dans le répertoire **${CATALINA_HOME}/webapp/axis/WEB-INF/classes**

Le contenu du fichier (`DeployHello.wsdd`) de déploiement de notre exemple `HelloService.java` est le suivant :

```
<!-- Inclure les espaces de noms -->

<deployment

xmlns="http://xml.apache.org/axis/wsdd/"

xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

<service name="HelloService" provider="java:RPC">

     <parameter name="className" value="HelloService"/>

     <parameter name="allowedMethods" value="*"/>

</service>
</deployment>
```

Un peu d'explications de ce fichier, même si je vous conseille dans un premier temps de ne pas trop se focaliser sur les détails. Néanmoins, pour avoir plus de détails, télécharger un tutoriel sur Axis.

L'élément XML `<deployment>` inclut les espaces de noms (pour éviter les ambiguïtés, voir cours) nécessaires. L'élément `<service>` spécifie les détails du service à déployer.

Dans l'élément `<service>` précédent, `name` spécifie l'URI du service déployé et `provider` désigne le `handler` qui appelle l'objet réel de la classe du service. Dans notre exemple, nous utilisons `java:RPC`, qui est de la class `org.apache.axis.providers.RPCProvider`, comme fournisseur. Il existe d'autre fournisseur comme `java:MSG` pour les services messageries.

Dans l'élément `<service>`, nous spécifions les paramètres en utilisant la balise `<parameter>`. `ClassName` sert à spécifier la class Java qui fournit le service souhaité. Le nom de la classe est donné par la valeur `value` du paramètre. Le fichier .class de cette class doit se trouver dans le répertoire ${CATALINA_HOME}/webapps/axis/WEB-INF/classes. C'est à dire HelloService.class est à mettre dans ce répertoire.

Ensuite nous spécifions, le paramètre `AllowedMethods`. Celui-ci indique les méthodes à exposer au monde extérieur (c'est bien une personnalisation**)**. Le symbole * dans le champ `value` indique que toutes les méthodes sont exposées au monde extérieur. Il est également possible de spécifier une liste de méthodes séparées par des espaces, voir l'exemple du service `CalculService.java`.

Déploiement
Après avoir écrit le fichier de déploiement, il faut réaliser le déploiement effectif en utilisant le fichier WSDD ainsi décrit. Pour cela on utilise l'utilitaire `AdminClient`.

`AdminClient` est un utilitaire fournit par Axis qui lit le fichier WSDD et déploie le service Web. Pour le lancer il suffit de taper au niveau shell :

```
$ java org.apache.axis.client.AdminClient DeployHelloService.wsdd
```

On doit avoir apparaître les messages :

```
-Processing file DeployHelloService.wsdd
<Admin> Done Processing </Admin>
```

Vous pouvez constater que ce service s'affiche effectivement dans la liste des services web déployés : http://localhost:8080/axis/servlet/AxisServlet

Tester le déploiement : http://localhost:8080/axis/services/HelloService, découvrez la description wsdl : http://localhost:8080/axis/services/HelloService?wsdl

Retirer le service (undeploy) :

Développement d'un client Java

Il suffit de changer dans les exemples de client JAVA précédent:

```
String endpoint ="http://localhost:8080/axis/HelloService.jws";

par

String endpoint ="http://localhost:8080/axis/services/HelloService";

Ou par

String endpoint ="http://localhost:8080/axis/servlet/AxisServlet";
call.setOperationName(new QName("HelloService","sayHello"));
```

Et de le compiler et exécuter.

Je résume cette approche: l'écriture de la classe de service `HelloService.java`, la compiler et la placer dans `${CATALINA_HOME}/webapps/axis/WEB-INF/classes`, écriture du fichier de déploiement `DeployHelloService.wsdd`, lancer l'utilitaire `AdminClient`, écrire un client Java comme dans le cas JWS avec un changement de de l'URL endpoint ou de l'URL endpoint et le paramètre call.setOperationName()

Un autre Exemple avec WSDD

On va utiliser l'exemple `CalculService.java`. Compiler ce fichier et placer la classe dans `${CATALINA_HOME}/axis/WEB-INF/classes`.

Le fichier DeployCalculService.wsdd

```
<!-- Inclure les espaces de noms -->
<deployment            xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<service name="CalculService" provider="java:RPC">
     <parameter name="className" value="CalculService"/>
     <parameter name="allowedMethods" value="*"/>
</service>
</deployment>
```

```
Ou par exemple si l'on veut exposer uniquement somme
<!-- Inclure les espaces de noms -->
<deployment            xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
<service name="CalculService" provider="java:RPC">
     <parameter name="className" value="CalculService"/>
     <parameter name="allowedMethods" value="somme"/>
</service>
</deployment>
```

`AdminClient`

```
$ java org.apache.axis.client.AdminClient DeployCalculService.wsdd
```

Client Java

Il suffit de changer dans les exemples de client JAVA précédent l'URL endpoint par:

```
String endpoint ="http://localhost:8080/axis/services/CalculService";

Ou   par

String endpoint ="http://localhost:8080/axis/servlet/AxisServlet";
call.setOperationName(new QName("CalculService","somme"));
```

Et de le compiler et exécuter.

IV) Utilisation de l'utilitaire WSDL avec Axis

WSDL (document XML) décrit l'interface exposée par un service Web. Il décrit les

différentes méthodes Web exposées par le service Web, ainsi que leurs paramètres et types retournés.

Axis supporte WSDL de trois manières :

a) Après avoir déployé le service, un document WSDL automatiquement généré peut être obtenu en accédant à l'URL du service à partir d'un navigateur Web standard et en attachant wsdl à la fin de l'URL. (exemple **http://localhost:8080/axis/Carre.jws?wsdl**)
b) L'outil **WSDL2Java** construit des proxies et des structures de services avec des descriptions WSDL.
c) Un outil **Java2WSDL** construit WSDL à partir de classes java.

IV-1) Utilisation de WSDL2Java

Le but est de créer un client par le bias des stubs (proxy) que l'utilitaire WSDL2Java va générer.

On va construire un service Web qui expose une méthode carre() qui retourne le carre d'un entier passé en paramètre. La classe Java de cet exemple est la suivante :

```
public class  Carre{
Public int carre(int a) { return a*a;}
}
```

Sauvegardons cette classe Carre.java dans ${CATALINA_HOME}/webapps/axis/ en la renommant Carre.jws.

On peut ainsi récupérer son WSDL comme auparavant (http://localhost:8080/axis/Carre.jws?wsdl). Nommant ce fichier `Carre.wsdl`. Dans un scénario réel, l'éditeur du service Web peut placer ce fichier dans un annuaire.

Pour générer les classes proxy à partir de ce fichier WSDL, nous utilisons la commande suivante:

```
$ java org.apache.axis.wsdl.WSDL2Java -p  repertoire Carre.wsdl.
```

Quatre fichiers seront générés dans le package repertoire : `Carre_PortType.java`, `CarreService.java`,`CarreServiceLocator.java`,`CarreSoapBindingStub.java`

`Carre_PortType.java` est une interface qui héberge les signatures des méthodes Web exposées par le service Web. Voici son code:

```
/**
 * Carre_PortType.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package reperoire;

public interface Carre extends java.rmi.Remote {
    public int carre(int a) throws java.rmi.RemoteException;
}
```

Rappelez-vous les RMIs !!!!!!!!!!!!!!

CarreService est une interface qui contient les signatures des méthodes utilisées pour localiser le service. Le code de l'interface est le suivant :

```
/**
 * CarreService.java
 *
 * This file was auto-generated from WSDL
 * by the Apache Axis WSDL2Java emitter.
 */

package repertoire;

public interface CarreService extends javax.xml.rpc.Service {
    public java.lang.String getCarreAddress();

    public repertoire.Carre_PortType getCarre() throws
javax.xml.rpc.ServiceException;

    public repertoire.Carre_PortType getCarre(java.net.URL portAddress)
throws javax.xml.rpc.ServiceException;
}
```

Les deux méthodes acquéreurs retournent la référence au serveur. La première méthode ne prend aucun argument, alors que la seconde accepte l'URL du service comme argument.

La classe CarreServiceLocator implémente l'interface CarreService et fournit donc l'implémentation des deux méthodes acquéreurs.

La classe CarreSoapBindingStub implémente l'interface Carre_PortType. Cette classe définit l'objet Call et définit les paramètres nécessaires pour la méthode Web spécifiée.

Les codes des classes CarreServiceLocator et CarreSoapBindingStub sont trop grands pour l'inclure dans le document.

Création d'un client par le biais de stubs générés :

Il est maintenant simple d'écrire un programme qui utilise les classes que nous venons de générer. Voici son code :

```
// Un client TestCarre qui utilise les proxys générés par
//WSDL2Java
// Rappel : J'ai récupéré d'abord Carre.wsdl (comme d'habitude)
// J'ai lancé java org.apache.axis.wsdl.WSDL2Java –p repertoire
// Carre.wsdl

// Ca m'a généré un package repertoire qui contient
// 1) Carre_PortType.java contient une interface de proxy du service Web.
// des méthodes exposées par le serveur.
// 2) CarreService.java est une interface qui contient les signatures des
//méthodes utilisées pour localiser le serveur Web.
// Deux méthodes qui retournent la référence au serveur. La
//première ne prend aucun argument. Le deuxième accepte l'URL du service
//comme argument
```

```
//    3)   la   classe   CarreServiceLocator   implémente   l'interface
//CarreService.java
//   La   classe   CarreSoapBindingStub.java   implémente   l'interface
//CarrePortType.java. //Cette class crée  l'objet Call et définit les
//paramètres  nécessaires pour la méthode Web spécifiée.


import repertoire.*;

public class TestCarre{
public static void main (String [] args)
{
   // créer une instance sur la classe de localisation
      CarreServiceLocator  locator= new CarreServiceLocator();

try {
      Carre_PortType server =locator.getCarre();

      int   a = server.carre(4);

       System.out.println("Resulat "+ a);
}
   catch(java.lang.Exception e)
 { e.printStackTrace();}

}
}
```

Un deuxième exemple. Récupérer le wsdl de CalculService.java
La classe de ce service

```
public public CalculService{
public int somme (int a, int b){return a+b;}
public int produit (int a, int b){return a*b;}
public int difference (int a, int b){return a-b;}
}

$ java org.apache.axis.wsdl.WSDL2Java –p repertoire CalculService.wsdl

// La client

/* Ce fichier je l'ai créer pour teste WSDL2Java
J'ai récupérer d'abord CalculService.wsdl
 j'ai lancé  java org.apache.axis.wsdl.WSDL2Java -p repertoire
CalculService.wsdl
Ca m'a généré un package appelé repertoire

 Ce package contient:
 1) CalculService_PortType.java contient une interface de proxy du service
Web. Le proxy héberge  des méthodes exposées par le serveur.

 2) CalculServiceService.java. est une interface qui contient les
signatures des méthodes utilises pour localiser le serveur Web.
 Deux méthodes qui retournent la référence au serveur. La première ne
prend  aucun argument. Le deuxième accepte l'URL du service comme argument


3) la classe CalculServiceServiceLocator implémente l'interface
CalculServiceService
```

 4) La classe CalculServiceSoapBindingStub.java implémente l'interface CalculService.java. Cette class Cree l'objet Call et définit les paramètres nécessaires.

 j'ai mis dans classpath jusqu'au pere de repertoire
 On import les classe du repertoire java est capabable d'aller jusqu'au dossier père de repertoie mais après il faut importer les classes

```
import repertoire.*;

public class TestCalculService{
public static void main (String [] args)
{
// créer une instance sur la classe de localisation
// Pour pouvoir localiser le serveur Web

    CalculServiceServiceLocator locator= new CalculServiceServiceLocator();

try {
    // retourne la référence du serveur Web

      CalculService  server =locator.getCalculService();
// j'appelle la méthode somme
      int   a = server.somme(4,5);
       System.out.println("somme "+ a);

// j'appelle la méthode produit
int b= server.produit(4,5);
 System.out.println("produit "+ b);

  // j'appelle la méthode soustraction
   int c= server.soustraction(4,5);
       System.out.println("soustraction "+ c);

}
   catch(java.lang.Exception e)
 { e.printStackTrace();}

}
}
```

VI-2) Utilisation de Java2WSDL

L'utilitaire Java2WSDL permet de générer WSDL à partir de la classe. Considérons l'exemple déjà étudié Carre.java.

a) # javac Carre.java
b) # java org.apache.axis.wsdl.Java2WSDL -o Carre.wsdl -l http://localhost:8080/axis/services/Carre -n "urn:Carre" Carre

Génère le fichier Carre.wsdl et place le service à la place indiquée.

La liste suivante donne la signification de chaque option utilisée :
-o: Nom de fichier WSDL
-l: Emplacement du service
- n: Espace de noms cible du fichier WSDL.

Visualiser le fichier produit Carre.wsdl

V) Apache XML-RPC

Mise en garde, ce domaine évolue très rapidement, les classes utilisées doivent être réactualisées à partir du site : http://ws.apache.org/xmlrpc/

XML-RPC est un protocole RPC basé sur XML. Il s'agit d'une spécification et d'un jeu d'implémentation qui permet aux logiciels s'exécutant sur des systèmes disparates, dans des environnements différents, d'effectuer des appels de procédures sur Internet. Cela conduit à effectuer des appels de procédures distants via HTTP POST pour ce qui est du transport et XML pour ce qui est du codage. XML-RPC a été conçu pour être aussi simple que possible, tout en autorisant les transmissions, le traitement et les retours de structures de données complexes. Une procédure s'exécute sur le serveur et la valeur qu'elle retourne est également formatée en XML. Les paramètres de procédures peuvent être des scalaires, des nombres, des chaînes, des dates, ainsi que des structures d'enregistrements et de listes.

Apache XML-RPC est une implémentation Java de XML-RPC. Elle propose la classe org.apache.XmlRpc qui permet de créer des applications clients en utilisant les appels XML-RPC sur un serveur distant. Un client peut appeler les méthodes distantes de manières synchrone ou asynchrone. Nous illustrons ces deux méthodes. Pour commencer télécharger XML-RPC à partir du web http://ws.apache.org/xmlrpc/download.html et installer ce logiciel et mettre à jour votre CLASSPATH.

Je vous conseille avant de commencer à programmer les deux exemples suivants de consulter la page http://ws.apache.org/xmlrpc/apidocs/index.html

1) Premier Exemple de Service Web

Le premier service web permet d'exposer une méthode sumAndDifference qui prend deux entiers en paramètres et retourne la somme et la différence.

```
import java.util.Hashtable;
import org.apache.xmlrpc.WebServer;

public class ServeurSomDiff {
            public ServeurSomDiff () {
   // C'est un objet ordinaire. Il peut avoir un constructeur //et des
variables. Les méthodes public sont exposées aux // clients.
          }

    // On a besoin de Hashtable car deux valeurs de retours
// La méthode exposée du serveur
        public Hashtable sumAndDifference (int x, int y) {
            Hashtable result = new Hashtable();
            result.put("somme", new Integer(x + y));
            result.put("difference", new Integer(x - y));
            return result;
        }

        public static void main (String [] args) {
            try {
               // Invoke me as <http://localhost:8000/RPC2>.
                  //On crée un objet server qui recevra les requêtes pour
// notre web service, il écoute le port 8000.
```

```
                    WebServer server = new WebServer(8000);

        // on lie la chaîne SOMDIFF au serveur ServeurSomDiff()
          server.addHandler("SOMDIFF", new ServeurSomDiff());

// On lance le serveur
        server.start();

      System.out.println("Serveur lance sur http://localhost:8000/RPC2");

    } catch (Exception exception)
{System.err.println("JavaServer: " + exception.toString());
               }
          }
  }
```

Commentaire :
La méthode main() crée une instance de la classe `WebServer` qui écoute le port 8000 dans l'attente de requêtes clientes et crée un handler appelé « SOMDIFF » que les clients utilisent pour appeler une méthode sur le serveur. La classe `WebServer` est fourni par la package `XML-RPC org.apache.xmlrpc` que nous avons importé.
Compiler et exécuter le serveur.

On va maintenant écrire un client Java. Deux manières : un client synchrone, ou un client asynchrone.

Un client synchrone.

```
// Un client Synchrone bloqué jusqu'a ce que le serveur retourne le
//résultat


    import java.util.Vector;
    import java.util.Hashtable;
    import org.apache.xmlrpc.*;

    public class SynClientSomDiff {

        // serveur_url l'URL de localisation du serveur


 private final static String server_url = "http://localhost:8000/RPC2";
        public static void main (String [] args) {
            try {


                      // Création d'un objet représentant le serveur.
               XmlRpcClient server = new XmlRpcClient(server_url);


// Construction de la liste des paramètres
               Vector params = new Vector();
               params.addElement(new Integer(5));
               params.addElement(new Integer(3));


// Appel du serveur et réception du résultat
```

```
Hashtable result=(Hashtable)  server.execute  ("SOMDIF.sumAndDifference", params);

// Récupération du résultat à partir du Vector

int sum = ((Integer) result.get("somme")).intValue();

int difference = ((Integer) result.get("difference")).intValue();

// Affichage des résultats
System.out.println("Somme: " + Integer.toString(sum) +

System.out.println("Difference: " + Integer.toString(difference));

            }
catch (XmlRpcException exception) {
System.err.println("JavaClient: + Integer.toString(exception.code) + ": " +
                                   exception.toString());
            } catch (Exception exception) {
                System.err.println("JavaClient: " + exception.toString());
            }
        }
    }
```

Commentaire :
Notez que nous avons spécifié `l'URL` du serveur comme paramètre du constructeur de classe `XmlRpcClient`.

Ensuite, nous créons un objet `Vector` qui va stocker les paramètres à passer aux méthodes de class du serveur. L'un des types de paramètres de la méthode du serveur `execute()` doit être un `Vector`.

La méthode `sumAndDifference`() du serveur est appelé via la méthode `execute()`. Cet appel indique au serveur le nom de la méthode comme premier paramètre et sa liste de paramètres (voilà pourquoi nous avons employé un `Vector`) comme second paramètre. La valeur retournée par la méthode du serveur est assignée à une variable `Hastable`.

Un client asynchrone

```
// Un client asynchrone qui n'attend pas le retour  du serveur

   import java.util.Vector;
    import java.util.Hashtable;
    import org.apache.xmlrpc.*;

    public class AsynClientSomDiff {

   private final static String server_url = "http://localhost:8000/RPC2";

     public static void main (String [] args) {
            try {
              XmlRpcClient server = new XmlRpcClient(server_url);
 Vector params = new Vector();
                params.addElement(new Integer(5));
                params.addElement(new Integer(3));

       server.executeAsync("sample.sumAndDifference", params, new AsynCallHandlerSomDiff());
```

```
/    / des * sont affichées en attendant le résultat
   for (int i=0; i<2000; i++) System.out.print(*.");

            } catch (Exception exception) {
                System.err.println("JavaClient: " + exception.toString());
            }
        }
    }
```

2) Deuxième exemple

Nous allons développer une application serveur simple qui retourne au client un prix d'action correspondant au code de l'action (symbole). Pour que cette application reste simple, nous avons fait appel à une table de hachage `Hashtable` qui contient les symboles boursiers et leurs prix. Voici le code complet du programme :

```
import java.util.Hashtable;

import org.apache.xmlrpc.*;


 public class ServeurQuote {
        Hashtable StockesQuotes;
 // le constructeur de cette classe initialise cette Hashtable
public ServeurQuote () {
          StockesQuotes = new Hashtable ();
          StockesQuotes.put ("SUN", new Double(120.00));
          StockesQuotes.put ("IBM", new Double(1460.00));
          StockesQuotes.put ("AIRFRANCE", new Double(1745.00));
        }

// La méthode GetStockPrice() prend un symbole boursier comme paramètre et
//retourne le cours actuel de l'action à l'appelant.

public  Double getStockPrice(String symbole)
{ return (Double) (StockesQuotes.get(symbole));
}
 public static void main (String [] args) {
 try {
     //le serveur doit être invoqué par http://localhost:8000/RPC2.
// Une instance de la classe WebServer est crée et écoute sur le port 8000
      WebServer server = new WebServer(8000);
// Un handler appelé STOCK est crée. Ce handler sera utilisé par le client
//pour appeler une méthode sur le serveur.
  server.addHandler("STOCK", new ServeurQuote());
// lancer le serveur
 server.start();

      System.out.println(" A invoquer sur http://localhost:8000/RPC2");

            } catch (Exception exception) {
                System.err.println("JavaServer: " + exception.toString());
            }
        }
    }
```

Un client synchrone

```
   import java.util.Vector;
    import java.util.Hashtable;
    import org.apache.xmlrpc.*;
    public class SynClientQuote {
    private final static String server_url = "http://localhost:8080/RPC2";

 public static void main (String [] args) {
      if (args.length!=1) { System.out.println("Usage=> java JavaClient2
<StockSymole>");
            System.exit(0);}

try { XmlRpcClient server = new XmlRpcClient(server_url);
 // Construction des paramètres
 Vector params = new Vector();
params.addElement(args[0]);
             Double price = (Double)
server.executeAsync("STOCK.getStockPrice",params);

 // DecimalFormat df = new DecimalFormat("##.00");
// afficher le resultat
System.out.println("Le prix est : "+ price.doubleValue()+"$");


}
catch (XmlRpcException exception)
{ System.err.println("JavaClient: XML-RPC Fault #" +
                                   Integer.toString(exception.code) + ": "
+
                                   exception.toString());
            } catch (Exception exception) {
                System.err.println("JavaClient: " + exception.toString());
            }
        }
    }
```

VI) AXIS2

Axis2 n'est pas une version améliorée d'axis1, mais une implémentation complétement différente de SOAP en Java.

Cette section n'est pas très détaillée, elle est donnée à titre indicatif. Il est par conséquent, conseillé de consulter le site officiel d'axis. Vous trouvez par ailleurs, cet aperçu complet sur beaucoup de sites.

Il s'agit de la version 1.5 d'axis. Cette version est complétement différente. Je vous donne ici quelques exemples, néanmoins pour plus de détails visiter le site http://axis.apache.org/axis2/java/core/

Installer axis2 dans le répertoire indiqué par $AXIS2_HOME. Le répertoire axis2 comporte des fichiers et des répertoires. Exemple :

$ls $AXIS2_HOME

bin   lib   repository   conf   samples   webapp

et des fichiers.

Le but de ce TP est de développer un premier exemple d'un service Web en utilisant Axis2.

<u>Première étape</u> :

On va développer le fameux exemple hello world. Pour cela, on va créer un répertoire nommé helloservice qui contient deux répertoires META-INF et le répertoire hello.

Dans le répertoire META-INF, on crée le fichier services.xml qui décrit le déploiement de service. Son contenu est le suivant :

```
<service name="HelloWorldService" scope="application">

          <description>
             Hello World  Service
          </description>

 <messageReceivers>
    <messageReceiver              mep=http://www.w3.org/2004/08/wsdl/in-only
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>

      <messageReceiver           mep="http://www.w3.org/2004/08/wsdl/in-out"
      class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>

</messageReceivers>

 <parameter name="ServiceClass">
             hello.HelloWorld
  </parameter>

</service>
```

Ces balises sont évocatrices et il est aisé d'appréhender l'intérêt et le but, mais dans un premier temps, il est conseillé de ne pas se focaliser sur ce détail.

Le répertoire hello contient la classe HelloWorld.java suivante:

```
public class HelloWorld{
     String sayHello(String argument)
     {return      "Hello World";
     }
```

Ce fichier HelloWordl.java est stocké sans compilation dans le répertoire hello.

Deuxième étape : Déploiement proprement dit du service.

Pour déployer le service ainsi développé il suffit Cette étape de copier le répertoire helloservice (il contient les deux répertoires hello et META-INF) au sein d'un fournisseur de services web. Dans notre cas, il s'agit d'axis2.

Si $AXIS2_HOME est le répertoire d'installation d'axis2, copier hello services dans $AXIS2_HOME/repository/services. Sous Linux il suffit :

cp -r helloservice ${AXIS2_HOME}/repository/services/

<u>Troisième étape</u> : Utilisation au service

Avant d'utiliser ce service, il faut au préalable lancer axis2 comme suit :

source axis2server.sh

#source est une commande shell et axis2service.sh se trouve dans le $AXIS2_HOME/bin, mais comme vous avez positionnez ce répértoire dans le PATH, il suffit donc de lancer la commande telle que indiqué.

Désormais le service HelloWorld est dans la liste des services de Axis2 accessibles sur la page dont l'URL est la suivante :

`http://localhost:8080/axis2/services/`

Si vous cliquez sur le lien du service **HelloWorldService**, vous verrez la définition WSDL (générée automatiquement par Axis2) de votre service web.

<u>Exécution du service</u>

Pour exécuter une méthode de votre service, il suffit d'indiquer l'URL suivante dans votre navigateur et on aura la réponse SOAP.

`http://localhost:8080/axis2/services/HelloWorldService/sayHello`

<u>Développement d'un client d'un service Web avec Axis2</u>

L'objectif de ce TP est d'utiliser/tester le service web développé dans l'exemple précédent. Nous allons développer un client Axis en utilisant les outils de génération automatique d'interface Java/XML fournis par cet outil. Nous utiliserons l'utilitaire WSDL2Java (voir les sections précédentes), qui permet de générer à partir de la description WSDL d'un service les différentes classes et interfaces clientes nécessaires à l'appel de ce service côté client.

<u>Etape 1</u>

Récupérer la description WSDL du l'exemple HelloWorld à l'adresse : http://localhost:8080/axis2/services/HelloWorldService?wsdl

On va maintenant générer les interfaces Java clientes du service HelloWorld avec la commande suivante :

 # wsdl2java.sh -uri http://localhost:8080/axis2/services/HelloWorldService?wsdl -d adb

Voir manuel d'utilisation de wsdl2java.

wsdl2java génère un certain nombre de classes et interfaces Java dont la classe HelloWorldServiceStub.java. Cette classe contient les différentes méthodes exposées par ce service. Au sens de la spécification WSDL, il s'agit d'un type de port (<<PortType>>) qui implémente plusieurs <<opérations>> (les méthodes de notre service). La classe HelloWorldServiceStub.java contient une classe pour chaque type de données complexe éventuellement défini par le service (dans le fichier .wsdl).

<u>Etape 2</u>

Avec les classes ainsi générées, voici à quoi ressemble le code du client.

```
import hello.HelloWorldServiceStub;
import hello.HelloWorldServiceStub.HelloResponse;
public class Client {
   public static void main (String [] args){
            try{
                //creation d'un stub pour le service Web Hello World
                HelloWorldServiceStub stub = new HelloWorldServiceStub();

                //invocation de la methode hello
                HelloResponse resp = stub.sayhello();
                // affichage de resultat
                System.out.println(resp.get_return());
            }catch(Exception e){
                System.out.println(e);
            }


        }
      }
```